China-pub.com
下载

第6章 测 试

在日常的手工或者通过办公机器的计算实践中，人们形成了一种习惯，那就是要检查计算的每个步骤。如果发现了错误，确认它的方式就是从第一次注意到错误的那个地方开始，做反向的处理。

Norbert Wiener(诺伯特·维纳)，《控制论》

测试和排错常常被说成是一个阶段，实际上它们根本不是同一件事。简单地说，排错是在你已经知道程序有问题时要做的事情。而测试则是在你在认为程序能工作的情况下，为设法打败它而进行的一整套确定的系统化的试验。

Edsger Dijkstra有一个非常有名的说法：测试能够说明程序中有错误，但却不能说明其中没有错误。他的希望是，程序可以通过某种构造过程正确地做出来，这样就不再会有错误了，因此测试也就不必要了。这确实是个美好的目标，但是，对今天的实际程序而言，这仍然还只是一个理想。所以在这一章里，我们还是要集中精力讨论如何测试，才能够更快地发现程序错误，工作更有成效，效率也更高。

仔细思考代码中可能的潜在问题是个很好的开端。系统化地进行测试，从简单测试到详尽测试，能帮我们保证程序在一开始就能正确工作，在发展过程中仍然是正确的。自动化能帮助我们减少手工操作，鼓励人们做更广泛的测试。在这方面也存在大量技巧，是设计程序的人们从实践中学到的。

产生无错误代码的一个途径是用程序来生成代码。如果某些程序的工作已经彻底理解清楚了，写代码的方式很机械，那么就应该把它机械化。用某种特定语言描述后，程序可以自动生成出来，这种情况也是很常见的。典型的例子如：我们把高级语言的程序编译成汇编代码；使用正则表达式来描述正文的模式；我们使用 `SUM(A1:A50)`一类的记法，表示的是在电子表格系统中一系列单元上的运算。对这些情况，如果所用的生成器或者翻译器是正确的，给出的描述完全正确，那么作为结果的程序必然也是正确的。我们在第 9章将对这个内涵丰富的领域做更细致的讨论。在本章中，我们将只对通过某些紧凑描述来生成测试实例的方法做一点简单讨论。

6.1 在编码过程中测试

问题当然是发现得越早越好。如果你在写代码时就系统地考虑了应该写什么，那么也可以在程序构造过程中验证它的简单性质。这样做的结果是，甚至代码还没有经过编译，就已经经过了一轮测试。这样，有些错误类根本就不会出现了。

测试代码的边界情况。一项重要技术是边界条件测试：在写好一个小的代码片段，例如一个循环或一个条件分支语句之后，就应该检查条件所导致的分支是否正确，循环实际执行的次数是否正确等。这种工作称为边界条件测试，因为你的检查是在程序和数据的自然边界上。例如，

China-pub.com
下载

应该检查不存在的或者空的输入、单个的输入数据项、一个正好填满了的数组，如此等等。这里要强调的观点是：大部分错误都出现在边界上。如果一段代码出毛病，毛病最可能是出在边界上。相反，如果它在边界上都工作得很好，一般来说在别的地方也能够这样。

下面的片段模拟fgets，它读入一些字符，直到遇到一个换行或者缓冲区满了：

```
?    int i;
?    char s[MAX];
?
?    for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i)
?        ;
?    s[--i] = '\0';
```

设想你刚刚写好这个循环，现在要在你的头脑里模拟它读入一行的过程。第一个边界测试很简单，读入一个空行，如果一个行的开始就是个换行符。我们很容易看到循环在第一次重复执行时就结束了，这时i被设置为0。随后，最后一行代码将i的值减小到-1，并把一个空字节写入s[-1]，这位于数组的开始位置之前。通过边界条件测试，错误被发现了。

如果我们重写这个循环，采用普通的用输入字符填充数组的惯用写法，它的形式应该像下面这样：

```
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n')
?            break;
?    s[i] = '\0';
```

重复前面的边界测试，对只有一个换行字符的行，很容易验证程序能正确处理： i是0，第一个输入字符跳出循环，而'\0'被存入s[0]中。对于在一个或两个字符之后是换行符的输入做类似检查，使我们相信在接近边界的地方这个循环也能工作。

当然，还有另一些边界条件需要检查。如果输入中有一个很长的行，或者其中没有换行符，保证i总是小于MAX-1的检测能处理这个问题。但如果输入为空那又会怎么样，这时对getchar的第一次调用就返回了EOF。我们必须增加新条件，检查这种情况：

```
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?            break;
?    s[i] = '\0';
```

边界检查可以捕捉到许多错误，但是未必是所有的错误。第 8章我们还要回到这个例子，在那里要说明，在这段代码里实际上还有一个移植性错误。

下一步应该是在另一端的边界上检查输入：检查数组接近满了、正好满了或者超过了的情况，特别是如果换行字符正好在这个时候出现。我们不准备在这里写出所有的细节，不过，这个例子是一个很好的训练。对于边界条件的思考将会提出一个问题：如果在 '\n'出现前缓冲区已满，程序应该做些什么？这是早在规范描述里就应该解决的问题，而测试边界条件能帮助我们识别它。

边界条件检查对发现“超出一个”的错误特别有效。在实践中，做这种检查已经变成了许多人的习惯。这样，大量的小错误在它们还没有真正发生前就已经被清除了。

测试前条件和后条件。防止问题发生的另一个方法，是验证在某段代码执行前所期望的或必须满足的性质(前条件)、执行后的性质(后条件)是否成立。保证输入取值在某个范围之内是前条件测试的一类常见例子，下面的函数计算一个数组里 n个元素的平均值，如果n小于或者等于0，就会有问题：

China-pub.com 下载

```
?    double avg(double a[], int n)
?    {
?        int i;
?        double sum;
?
?        sum = 0.0;
?        for (i = 0; i < n; i++)
?            sum += a[i];
?        return sum / n;
?    }
```

当n是0时avg应该返回什么？一个无元素的数组是个有意义的概念，虽然它的平均值没有意义。avg应该让系统去捕捉除零错误吗？还是终止执行？提出抱怨？或许是默默地返回某个无害的值？如果n是负数又该怎么办？这当然是无意义的但也不是不可能的。正像前面第 4章里建议的，按照我们的习惯，如果n小于等于0时或许最好返回一个0：

```
return n <= 0 ? 0.0 : sum/n;
```

当然，在这里并没有惟一的正确答案。

然而，忽略这种问题则一定是个错误的回答。在 1998年11月的《科学美国人》杂志上有一篇文章，描述了美国导弹巡洋舰约克敦号上的一起事故。一个船员错误地输入了一个 0作为数据值，结果造成了除零错，这个错误窜了出去，最后关闭了军舰的推进系统。约克敦号“死”在海上几个小时，就是因为某个程序没有对输入的合法性进行检查。

使用断言。C和C++在<assert.h>里提供了一种断言机制，它鼓励给程序加上前 /后条件测试。断言失败将会终止程序，所以这种机制通常是保留给某些特殊情况使用的，写在这里的错误是真正不应该出现的，而且是无法恢复的。我们可能在前面程序段里的循环前面加一个断言：

```
assert(n > 0);
```

如果这个断言被违反，它就会导致程序终止，并给出一个标准的信息：

```
Assertion failed: n > 0, file avgtest.c, line 7
Abort(crash)
```

断言机制对于检验界面性质特别有用，因为它可以使人注意到调用和被调用之间的不一致性，并可以进一步指出麻烦究竟是出在哪里。如果关于 n大于0的断言在函数调用时失败，它实际上就指明了调用程序是造成麻烦的根源，问题不在 avg本身。如果一个界面做了些修改，而我们忘记对那些依赖于它的程序做更新，断言使我们有可能在错误还没有造成实际损害之前把它抓住。

防御性的程序设计。有一种很有用的技术，那就是在程序里增加一些代码，专门处理所有“不可能”出现的情况，也就是处理那些从逻辑上讲不可能发生，但是或许 (由于其他地方的某些失误)可能出现的情况。前面在avg里加上检查数组长度是否为0或者负数就是一个例子。作为另一个例子，一个处理学生成绩的程序应该假定不会遇到负数或者非常大的数，但是它却应该检查：

```
if (grade < 0 || grade > 100)    /* can't happen */
    letter = '?';
else if (grade >= 90)
    letter = 'A';
else
    ...
```

China-pub.com
下载

这些都是防御性程序设计的实例：设法使程序在遇到不正确使用或者非法数据时能够保护自己。空指针、下标越界、除零以及其他许多错误都可以提前检查出来，或者是提出警告，或者是使其转向。防御性程序设计(这里并不想作为双关语[⊖])应该能很好地捕捉到约克敦号上的除零错误。

检查错误的返回值。一个常被忽略的防御措施是检查库函数或系统调用的返回值。对所有输入函数(例如fread或fscanf)的返回值一定要做检查，看它们是否出错。对文件打开操作(如fopen)也应该这样。如果一个读入操作或者文件打开操作失败，计算将无法正确地进行下去。

检查输出函数(例如fprintf或fwrite)的返回值也可以发现一些错误，例如，要向一个文件写入，而磁盘上已经没有空间了。检查 fclose的返回值也是有道理的，它返回 EOF说明操作过程中出了错，否则就返回 0。

```
fp = fopen(outfile, "w");
while (...)                    /* write output to outfile */
    fprintf(fp, ...);
if (fclose(fp) == EOF) {       /* any errors? */
    /* some output error occurred */
}
```

输出错误也可能成为严重问题。如果当时正在写的是一个贵重文件的新版本，这个检查就可能防止你在新文件并没有成功建立的情况下删掉老的文件。

在编程的过程中测试，其花费是最小的，而回报却特别优厚。在写程序过程中考虑测试问题，得到的将是更好的代码，因为在这时你对代码应该做些什么了解得最清楚。如果不这样做，而是一直等到某种东西崩溃了，到那时你可能已经忘记了代码是怎样工作的。即使是在强大的工作压力下，你也还必须重新把它弄清楚，这又要花费许多时间。进一步说，这样做出的更正往往不会那么彻底，可能更脆弱，因为你唤回的理解可能不那么完全。

练习6-1 对下面例子的边界情况做各种检查，然后根据需要按照第 1章的风格和本章的建议对它们进行修改。

(a) 这个函数想计算阶乘：

```
int factorial(int n)
{
    int fac;
    fac = 1;
    while (n--)
        fac *= n;
    return fac;
}
```

(b) 这个程序段是要把一个字符串里的字符按一行一个的方式输出：

```
i = 0;
do {
    putchar(s[i++]);
    putchar('\n');
} while (s[i] != '\0');
```

(c) 这个函数希望从src复制字符串到dest：

```
void strcpy(char *dest, char *src)
{
```

⊖ 防御性程序设计(defensive programming)。由于军舰有防御问题， programming有做方案、做规划的意思，这个词组又可以解释为防御计划，防御方案。作者这里说的是俏皮话。——译者

China-pub.com 下载

```
        int i;

        for (i = 0; src[i] != '\0'; i++)
            dest[i] = src[i];
    }
```

(d) 这是另一个字符串复制，要从s拷贝n个字符到t：

```
    void strncpy(char *t, char *s, int n)
    {
        while (n > 0 && *s != '\0') {
            *t = *s;
            t++;
            s++;
            n--;
        }
    }
```

(e) 一个数值比较：

```
    if (i > j)
        printf("%d is greater than %d.\n", i, j);
    else
        printf("%d is smaller than %d.\n", i, j);
```

(f) 一个字符类检查：

```
    if (c >= 'A' && c <= 'Z') {
        if (c <= 'L')
            cout << "first half of alphabet";
        else
            cout << "second half of alphabet";
    }
```

练习6-2 当我们在1998年末写这本书的时候，2000年问题逐渐显现出来了，这可能是有史以来最大的边界条件问题。

(a) 你应该对哪些日期做检查，以验证一个系统是否可能在2000年里工作？假定测试的执行是非常昂贵的，你在试过2000年1月1日之后，准备以什么顺序继续你的测试？

(b) 你怎样测试标准函数ctime，它返回一个具有下面形式的表示日期的字符串：

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

假定你的程序调用ctime，你怎样写你的代码，使它能够抵御一个有毛病的标准函数实现。

(c) 说明你怎样测试一个以下面形式打印输出日历的程序：

```
    January 2000
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

(d) 你认为在你所使用的系统里还有哪些时间边界？可能通过怎样的测试弄清楚它们是否能得到正确处理？

6.2 系统化测试

最重要的是应该系统化地对程序进行测试，这样你可以知道每一步测试的是什么，希望

China-pub.com
下载

得到什么结果。你应该有秩序地做这些工作，而没有忽略掉任何东西；应该做一个记录，以便能随时了解已经完成了哪些工作。

以递增方式做测试。测试应该与程序的构造同步进行。与逐步推进的方式相比，以“大爆炸”方式先写出整个程序，然后再一古脑地做测试，这样做困难得多，通常也要花费更长时间。写出程序的一部分并测试它，加上一些代码后再进行测试，如此下去。如果你有了两个程序包，它们已经都写好并经过了测试，把它们连接起来后就应该立即测试，看它们能否在一起工作。

例如，当我们写第4章的CSV程序时，第一步是写出刚好能读输入的那么多代码，这使我们能检查输入处理的正确性。下一步是写出把输入行在逗号处切开的代码。当这些部分都能工作后，我们再转到带引号的域，这样逐步工作下去，直到测试完所有的东西。

首先测试简单的部分。递增方式同样适用于对程序性质的测试。测试应该首先集中在程序中最简单的最经常执行的部分，只有在这些部分能正确工作之后，才应该继续下去。这样，在每个步骤中你使更多的东西经过了测试，对程序基本机制能够正确工作也建立了信心。通过容易进行的测试，发现的是容易处理的错误。在每个测试中做最少的事情去发掘出下一个潜在问题。虽然错误可能是一个比一个更难触发，但是可能并不更难纠正。

本节里我们要谈的是怎样选择有效的测试，以及按什么顺序去执行它们。在随后两节里我们将讨论如何将这种过程机械化，以使得它能高效地进行。测试的第一步，至少对于小程序或独立函数而言，是做一些扩展的边界条件测试 (在前面一节里已经讨论过)：系统化地测试各种小的情况。

假设我们有一个函数，它对一个整数数组做二分检索。我们将着手做下面的测试，按复杂性递增的顺序安排：

- 检索一个无元素的数组。
- 检索一个单元素的数组，使用一般的值，它
  - 小于数组里的那个元素；
  - 等于那个元素；
  - 大于那个元素。
- 检索一个两元素的数组，使用一般的值，这时
  - 检查所有的五种可能情况。
- 检索有两个重复元素的数组，使用一般的值，它
  - 小于数组里的值；
  - 等于数组里的值；
  - 大于数组里的值。
- 像检索两个元素的数组那样检索三个元素的数组。
- 像检索两个和三个元素的数组那样检索四个元素的数组。

如果函数能够平安地通过所有这些测试，它很可能已经完美无缺了。但是还需要对它做进一步的测试。

上面这个测试集非常小，完全能以手工方式进行。但还是最好为此建立一个测试台，以使测试过程机械化。下面的驱动程序是我们按尽可能简单的方式写出的，它读入一个输入行，其中包含要检索的关键码和一个数组大小。它建立一个具有指定大小的数组，其中包含值 1,

China-pub.com 下载

3, 5, …, 并在这个数组里检索指定的关键码。

```
/* bintest main: scaffold for testing binsearch */
int main(void)
{
    int i, key, nelem, arr[1000];

    while (scanf("%d %d", &key, &nelem) != EOF) {
        for (i = 0; i < nelem; i++)
            arr[i] = 2*i + 1;
        printf("%d\n", binsearch(key, arr, nelem));
    }
    return 0;
}
```

这当然是过分简单化了，但它也说明一个有用的测试台不必很大。很容易对这个程序做些扩充，使之可以执行更多的测试，而且需要更少的手工干预。

弄清所期望的输出。对于所有测试，都必须知道正确的答案是什么，如果你不知道，那么你做的就是白白浪费自己的时间。这件事看起来很明显，因为对许多程序而言，了解它们能否工作是很容易的事。例如，一个文件的拷贝或者是个拷贝，或者就不是。由一个排序程序得到的输出或者是排好序的或者不是，当然它还必须是原始输入的一个重排。

但是，也有许多程序的特性更难以说清楚，例如编译程序 (其输出是输入的一个正确翻译吗？)，数值算法(答案是不是在可容忍的误差范围之内？)，图形系统(所有的像素都在正确的位置上吗？)等等。对于这些系统，通过把输出与已知的值进行比较，用这种方式来验证它们就特别重要。

- 测试一个编译程序，方法应该是编译并运行某些程序文件。这些程序本身应该能产生输出，这样就可以用它们的输出和已知结果进行比较。
- 测试一个数值程序，应该产生那种能揭示算法边界情况的测试实例，既有简单的也有困难的。如果可能的话，写一些代码去验证输出特性的合理性。例如，对数值积分程序的输出可以检查其连续性，验证它的结果与闭形式积分解的相符情况。
- 要测试一个图形程序，仅看它能否画一个方框是不够的。应该从屏幕上把方框读回来，检查其边界是否正好位于它应该在的位置。

如果一个程序有逆计算，那么就检查通过该逆计算能否重新得到输入。例如加密操作和解密操作是互逆的，如果你加密的什么东西不能解密，那么一定有某些东西存在毛病。与此类似，无损的压缩和展开程序之间也是互逆的。把文件捆绑起来的程序也必须能分毫不差地把它们提取出来。有时存在着完成逆操作的多种方式，应该检查它们的各种组合情况。

检验应保持不变的特征。许多程序将保持它们输入的一些特征。有些工具，如 wc(计算行、词和字符数)和sum(计算某种检验和)可用于验证输出是否与输入具有同样大小、词的数目是否相同、是否以某种顺序包含了同样的字节以及其他类似的东西。另外还有一些程序可以比较文件的相等(cmp)或报告差异(diff)等。这些程序，或其他类似的东西在许多环境里都可以直接使用，也是非常值得用的。

一个字节频度程序可以用来检查数据特征的不变性，也可以报告出现了反常情况。例如，在假定只包含文本的文件里出现了非文本字符。下面就是一个这种程序，我们称其为 freq：

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
```

China-pub.com
下载


```
unsigned long count[UCHAR_MAX+1];

/* freq main: display byte frequency counts */
int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        count[c]++;

    for (c = 0; c <= UCHAR_MAX; c++)
        if (count[c] != 0)
            printf("%.2x  %c  %lu\n",
                c, isprint(c) ? c : '-', count[c]);
    return 0;
}
```

对于那些应该保持不变的特征，实际上也可以在程序内部进行检查。一个求数据结构的元素个数的函数就提供了一种简单的一致性检查能力。一个散列表应该有这样的性质：每个插入其中的元素都可以提取出来。这个条件很容易检查，只要有一个能把散列表内容转入一个文件或者数组里的函数。在任何时刻，插入一个数据结构的元素数目减掉删除的元素数目，必然等于结果里包含的元素数目，这是一个很容易检查的条件[⊖]。

比较相互独立的实现。一个库或者程序的几个相互独立的实现应该产生同样的回答。例如，由两个编译程序产生的程序，在相同机器上的行为应该一样，至少在大部分情况下应该是这样。

有时一个回答可以由两条完全不同的途径得到，或许你可以写出一个程序的某种简单版本，作为一个慢的但却又是独立的参照物。如果两个相互无关的程序得到的结果完全相同，这就是它们都正确的一个很好的证据。如果得到的结果不同，那么其中至少有一个是错的。

作者之一某次和另一个人一起工作，为一种新机器做一个编译程序。在工作中他们把对编译程序产生的代码做排错的工作分成两块：一个人去写为目标机器产生指令的软件；另一个人写排错系统的反汇编程序。在这种做法之下，指令集合解释或实现中的任何错误都很难在这两个部分里重复出现。当编译程序错误地编码了一条指令时，反汇编程序立刻就能发现它。编译的所有前期输出都用反汇编程序进行处理，再用这个输出结果与编译程序自己的排错(汇编)输出进行比较。这种策略在实践中非常行之有效，很快就找出两部分中的不少错误。如果还遗留下什么障碍，那就是当两个人对机器系统结构描述中的某种歧义性正好做了同样的错误理解时，这时排错就比较困难了。

度量测试的覆盖面。测试的一个目标是保证程序里的每个语句在一系列测试过程中都执行过，如果不能保证程序的每一行都在测试中至少经过了一次执行，那么这个测试就不能说是完全的。完全覆盖常常很难做到，即使是不考虑那些“不可能发生”的语句。设法通过正常输入使程序运行到某个特定语句有时也是很困难的。

存在一些衡量覆盖面的商用工具。轮廓程序(Profiler)通常是编译系统套装工具中的一部分，它提供了一种方法，能计算出程序里每个语句执行的次数，由此指明在特定测试中达到的覆盖面。

我们测试了第3章的马尔可夫程序，综合使用了各种技术。本章的最后一节将详细描述这些测试。

练习6-3 说明你准备如何测试`freq`。

⊖ 这个条件还依赖于插入时对重复元素的处理方式，可能有一些附加条件。——译者

China-pub.com 下载

练习6-4 设计和实现另一个freq程序，它能够统计其他类型的数据值出现的频度，如32位整数或者浮点数等。你能够做一个这类程序，使它能够很好地处理各种类型吗？

6.3 测试自动化

以手工方式做大量测试既枯燥无味又很不可靠，因为严格意义上的测试总要涉及到大量的测试实例、大量的输入以及大量的输出比较。因此，测试应该由程序来做，因为程序不会疲劳，也不会疏忽。花点时间写一个脚本程序或者一个简单程序，用它包装起所有的测试是非常值得做的，这能使一个完整的测试集可以通过(文字或者图形)一个按键而得以执行。测试集运行起来越容易，你运行它的次数就会越多，也越不会跳过它(即使时间非常紧张)。我们在写这本书时，就为验证所有程序写了一个测试集，每次无论做了什么修改，我们都再次运行它，其中有些部分能在每次重新编译之后自动运行。

自动回归测试。自动化的最基本形式是回归测试，也就是说执行一系列测试，对某些东西的新版本与以前的版本做一个比较。在更正了一个错误之后，人们往往有一种自然的倾向，那就是只检查所做修改是否能行，但却经常忽略问题的另一面，所做的这个修改也可能破坏了其他东西。回归测试的作用就在这里，它要设法保证，除了有意做过的修改之外，程序的行为没有任何其他变化。

有些系统提供了很丰富的工具，以帮助实现这种自动化。脚本语言使我们能很方便地写一些短的脚本，去运行测试序列。在Unix上，像cmp或diff这样的文件比较程序可用于做输出的比较，sort可以把共同的元素弄到一起，grep可以过滤输出，wc、sun和freq对输出做某些总结。利用所有这些很容易构造出一个专门的测试台。或许这些对于大程序还不够，但是对个人或者一个小组维护的程序则完全是适用的。

下面是一个脚本，它完成对一个名字为ka的应用程序的回归测试。它用一大堆各种各样的测试数据文件运行程序的老版本(old_ka)和新版本(new_ka)，对输出中每个不相同情况都给出信息。这个脚本是用Unix的shell写的，但是很容易改写为Perl或者其他脚本语言。

```
for i in ka_data.*          # loop over test data files
do
    old_ka $i >out1         # run the old version
    new_ka $i >out2         # run the new version
    if ! cmp -s out1 out2   # compare output files
    then
        echo $i: BAD        # different: print error message
    fi
done
```

测试脚本通常应该默不做声地运行，只在发生了某些未预见情况时才产生输出，就像上面所采用的方式。我们也可以在测试过程中打印每个文件名，如果什么东西出了毛病，就在文件名之后给出错误信息。这种显示工作进展情况的方式也能指出一些错误，例如无限循环，或者测试脚本没能运行正确的测试等。但是，如果测试运行得很好，喋喋不休的废话就会使人生厌。

上面用的-s参数能使cmp只报告工作状态，不产生输出。如果文件比较相等，cmp的返回状态为真，!cmp是假，这时就不打印任何东西。如果老的输出和新的不同，那么cmp将返回假，这时文件名和警告信息都会被输出。

China-pub.com
下载

回归测试实际上有一个隐含假定，假定程序以前的版本产生的输出是正确的。这个情况必须在开始时仔细进行审查，使这些不变性质能够一丝不苟地维持下去。如果在回归测试里潜伏着错误结果，人们将很难把它查出来，所有依赖它的东西将一直都是错的。所以，在实践中应该周期性地检查回归测试本身，以保证它的可靠性。

建立自包容测试。自包容测试带着它们需要的所有输入和输出，可以作为回归测试的一种补充。我们测试Awk的经验可能有些教益。我们对特定输入运行一些小程序，以测试各种语言结构，检查它们产生的输出是否正确。下面是为验证一个复杂增量表达式而写的一大堆测试里的一部分。这个测试运行一个 Awk的新版本(newawk)，让一个很短的Awk程序把输出写进文件里，用echo命令向另一个文件里写入正确输出，然后做文件比较，如果有差异就报告错误。

```
# field increment test: $i++ means ($i)++, not $(i++)

echo 3 5 | newawk '{i = 1; print $i++; print $1, i}' >out1

echo '3
4 1' >out2  # correct answer

if ! cmp -s out1 out2   # outputs are different
then
    echo 'BAD: field increment test failed'
fi
```

第一个注释是测试输入的一部分，它解释这个测试所做的到底是什么。

有时不用很多时间就可以构造出大量测试。对于简单表达式，我们可以建立一个小而特殊的语言，描述测试、输入数据以及所期望的输出。下面是一个不长的测试序列，考察在Awk里数值1可以用哪些方式表示：

```
try {if ($1 == 1) print "yes"; else print "no"}
1       yes
1.0     yes
1E0     yes
0.1E1   yes
10E-1   yes
01      yes
+1      yes
10E-2   no
10      no
```

第一行是被测试程序(所有出现在单词try后面的东西)，随后的每一行是一对数据，分别表示输入和我们所希望的输出，两者间用制表符分隔。第一个测试说明，如果第一个输入域是 1，那么输出应该是yes。前面7个测试都应该输出yes，而最后两个应该输出no。

我们用一个Awk程序(还能用其他什么东西呢？)把每个测试转换为一个完整的 Awk程序，然后再对各个输入运行它，并将实际输出与所希望的输出做比较。它只报告那些回答错误的情况。

类似机制也可以用来测试正则表达式匹配和代换命令。例如用一个写测试的小语言，可以很方便地写出许多测试情况，用程序来写程序去测试另一个程序，回报将是丰厚的 (第9章还有许多关于小语言和用程序来写程序的讨论)。

总计起来，我们为 Awk设计了大约一千个测试，而这整个的测试集用一个命令就可以运行，如果所有的东西都没有问题，那么测试将不产生任何输出。每当我们增加了一个新特征，

China-pub.com 下载

或者更正了一个程序错误，我们就在测试集里增加一些测试，验证有关操作的正确性。一旦程序做了修改，哪怕是最简单的，我们也会运行整个的测试集，这只不过花费几分钟时间。有时这种测试会发现某些完全没有预料到的错误，它也使 Awk的作者们多次避免了当众出丑。

如果你发现了一个程序错误，那么又该怎么办？如果这个错误不是通过已有的测试发现的，那么你就应该建立一个能发现这个问题的新测试，并用那个崩溃的代码版本检验这个测试。一个错误实际上可能会提出一批测试，甚至是需要检查的整整一集新问题，或许是要求在程序里增加一些防御机制，使程序能在其内部发现这个错误。

不要简单地把测试丢掉，因为它能够帮你确定一个错误报告是否正确，或是说明某些东西已经更正了。应保留所有关于程序错误、修改和更正的记录，它能够帮助你识别老问题、纠正新错误。在许多产业的程序设计公司里，这种记录是必须做的。对于你个人的程序设计，这也是一种小的投资，而它们的回报则会是源源不断的。

练习6-5 为printf设计一个测试集，尽可能多地使用机器的帮助。

6.4 测试台

到目前为止，我们讨论的主要是对独立程序以完全形式进行测试时所遇到的问题。这当然不是测试自动化的惟一形式，也不是在大程序构造过程中对程序部分做测试的方式，特别是如果你是程序组的成员。对于测试将要装进某些更大程序里的小部件而言，这也不是有效的方法。

要孤立地测试一个部件，通常必须构造出某种框架或者说是测试台，它应能提供足够的支持，并提供系统其他部分的一个界面，被测试部分将在该系统里运行。我们在本章前面用测试二分检索的小例子说明了一些问题。

如果要测试的是数学函数、字符串函数、排序函数以及其他类似东西，构造一个测试台通常很简单，这种测试台的主要工作是设置输入参数、调用被测试函数，然后检查结果。如果要测试一个部分完成的程序，构造测试台可能就是个大工作了。

为了展示这方面的情况，我们将用 memset(C/C++ 标准库里的一个mem... 函数)的测试来说明有关问题。 mem类函数通常是用汇编语言为特定机器写的，这是由于它们的执行性能非常重要。程序被调整得越仔细，也就越容易出错，因此也就越需要做彻底测试。

第一步应该是提供一个尽可能简单的 C版本，并保证它能够工作。这既可以作为检验执行性能的基准程序，更重要的是为了考察程序的正确性。在移到一个新环境时，应该总带着简单程序版本，一直使用它们，直到经过仔细调整的东西能工作为止。

函数memset(s, c, n)把存储器里从s开始的n个字节设置为c，最后返回s。如果不考虑速度，这个函数很容易写出来：

```
/* memset: set first n bytes of s to c */
void *memset(void *s, int c, size_t n)
{
    size_t i;
    char *p;

    p = (char *) s;
    for (i = 0; i < n; i++)
        p[i] = c;
    return s;
}
```

China-pub.com
下载

但是如果考虑速度，就必须用一些技巧，例如一次设置 32位或者64位的一个字。这样做可能引入程序错误，所以必须做范围广泛的测试。

测试采用在可能出问题的点上做穷尽检查和边界条件测试相结合的方式。对于 memset而言，边界情况包括*n*的一些明显的值，如0、1和2，但也应包括2的各个幂次以及与之相近的值，既有很小的值，也包括2^{16}这样的大数，它对应许多机器的一种自然边界情况，一个 16位的字。2的幂值得注意，因为有一种加速 memset的方法就是一次设置多个字节，这可以通过特殊的指令做；也可能采用一次存一个字的方式，而不是按字节工作。与此类似，我们还需要检查数组开始位置的各种对齐情况，因为有些错误与开始位置或者长度有关。我们将把作为目标的数组放在一个更大的数组里面，以便在数组两边各建立一个缓冲区域或安全边缘，这也使我们可以方便地试验各种对齐情况。

我们还需要对各种不同c值做检查，包括0、0x7F(最大的有符号数，假定采用的是 8位字节)、0x80、0xFF(检查与有符号字符或无符号字符有关的可能错误)以及某些比一个字节大的值(以保证使用的就是一个字节)。我们必须将存储区初始化为某种已知模式，与这些字符值都不同，这样才能查清memset是不是向合法区域之外写了东西。

我们可以用最简单方式为测试提供对照的标准：分配两个数组，在数组里对 n、c和偏移量在各种组合下的情况做比较：

```
big = maximum left margin + maximum n + maximum right margin
s0 = malloc(big)
s1 = malloc(big)
for each combination of test parameters n, c, and offset:
    set all of s0 and s1 to known pattern
    run slow memset(s0 + offset, c, n)
    run fast memset(s1 + offset, c, n)
    check return values
    compare all of s0 and s1 byte by byte
```

如果memset有问题，写到了数组界限之外，那么最可能受到影响的是靠近数组开始或结束位置的字节。我们预留下缓冲区域，以使被破坏的字节容易观察到，也使因程序错误复写掉程序其他部分的可能性变得小些。为检查函数是否写出了范围，我们需要比较 s0和s1的所有字节，而不仅是那n个应该写入的字节。

这样，一个合理的测试集可能应该包含以下内容的所有组合：

```
offset = 10, 11, ..., 20
c = 0, 1, 0x7F, 0x80, 0xFF, 0x11223344
n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
    31, 32, 33, ..., 65535, 65536, 65537
```

对从0到16的各个*i*，这里的n值至少包括了所有的2^i-1、2^i和2^i+1。

这些值不应该写进测试台的主要部分，但是应该出现在一个用手工或者程序建立的数组里。最好是自动地产生它们，因为这样才能方便地描述出更多 2的幂，或许包含更多的偏移量和更多的字符。

这些测试能给memset一个彻底表现的机会，而构造出它们，并让程序执行花费的时间并不多。对于上面的值，总共只有不到 3500个测试情况。这个测试完全是可移植的，在需要时可以把它搬到新的环境里。

作为一个警告，请读者注意下面的故事。有一次我们把 memset测试程序的一个拷贝给了某些人，他们正在为一个新处理器开发一个新的操作系统和一个函数库。几个月以后，我们

China-pub.com 下载

(作为原来测试程序的作者)开始使用这种机器，发现一个大应用程序在它自己的测试集上出了毛病。我们对程序做跟踪，最后把问题归结到memset的汇编语言实现中有一个与符号扩展有关的微妙错误。由于某些不清楚的原因，库的实现者修改了我们的 memset测试程序，不让它检查0x7F以上的c值。当然，当我们认识到memset非常可疑后，用原来的测试程序立刻就把这个错误揪出来了。

像memset这样的函数对于彻底测试非常敏感。它们很小，很容易证明测试集已经穷尽了代码中所有的可能执行路径，形成了一个完全覆盖。例如，要对 memmove做重叠、方向和对齐方式上所有组合的测试是完全可能的。这里所说的穷尽，并不是所有可能的复制操作，而是指所有的具有代表性的不同输入情况。

与其他所有测试方法一样，测试台方法也需要有正确答案来验证被测试的操作。这里的一种重要技术，正是我们在测试memset时使用的，就是用一个完全能相信是正确的简单版本与一个可能不正确的新版本做比较。这件事可以在测试的不同阶段进行，下面的例子也说明了这个问题。

作者之一实现过一个光栅图形程序库，其中需要提供的一个操作是从一个图像里复制一个像素块到另一个图像。根据参数不同，这个操作完成的可以是简单的存储区拷贝；或者需要把像素值从一个颜色空间转换到另一个颜色空间；或者它需要做“贴块” (tile)，也就是说重复地做输入的拷贝，铺遍一个矩形区域；还可以是这些特征和其他特征的组合。这个操作的规范很简单，而要想高效地实现，就需要写出大量处理各种情况的特殊代码。要保证所有的代码都正确，需要一种有效的测试策略。

首先，我们通过手工写出了最简单的代码，使之能正确完成对一个像素的操作。用它来测试库函数对一个像素的处理。一旦完成了这个阶段，库函数对单个像素的操作就成为可信任的了。

下一步，手工写出调用库函数的代码，每次操作一个像素，构造出一个很慢的程序，实现对一个水平行中所有像素的操作，并用它与库中效率更高的一行操作程序做比较。在这个工作完成后，我们认为库函数已经能完成对行的操作了。

这样继续下去，用行构造矩形，用矩形构造贴块，依此类推。在这个过程中发现了许多程序错误，包括测试程序本身的错误，但是这也正说明了该方法的有效性。我们同时测试了两个互相独立的实现，在此过程中建立了对两者的信任。如果一个测试失败，测试程序将打印出一个细致的分析，帮助了解什么东西可能出错，同时也可以检查测试程序本身的工作是否正确。

这个库在后来的许多年月里做过修改或移植，在这些工作中，该测试程序又反复地表现出它在发现程序错误方面的价值。

由于其一层一层进行的性质，每次使用这个测试程序都需要从零开始，逐步建立起这个程序本身对库的信任。附带地说，这个测试程序不是彻底的、而是概率性的，它生成随机的测试实例。但是，只要经过足够长时间的运行，它最终可能探查出代码中的所有错误。由于这里实际存在数目巨大的可能测试情况，与设法用手工方式构造完全的测试集相比，这种策略更容易实施，它又比做穷尽测试的效率高得多。

练习6-6 按照我们描述的方式为memset建立一个测试台。

练习6-7 为mem...函数族的其他函数建立测试台。

China-pub.com
下载

练习6-8 讨论对数值函数的测试方式，数值函数是指可以在 math.h里找到的如sqrt、sin等函数。哪些输入值是有意义的？可以执行哪些独立的检查？

练习6-9 为C标准库的str... 函数族的函数，如strcmp，定义有关的测试机制。其中的一些函数比mem...族的函数复杂得多，特别是strtok和strcspn这样的单词化函数，因此需要做更复杂的测试。

6.5 应力测试

采用大量由机器生成的输入是另一种有效的测试技术。机器生成的输入对程序的压力与人写的输入有所不同。量大本身也能够破坏某些东西，因为大量的输入可能导致输入缓冲区、数组或者计数器的溢出。这对于发现某些问题，例如程序对在固定大小存储区上的操作缺乏检查等，是非常有效的。人本身常常有回避“不可能”实例的倾向，这方面的例子如空的输入，超量级、超范围的输入、不大可能建立的特别长的名字或者特别大的数据值等等。与人相反，计算机会严格按照它的程序做生成，不会回避任何东西。

为说明这个问题，下面是Microsoft Visual C++ Version5.0编译程序产生的一个输出行，是在编译马尔可夫程序的 C++ STL实现时产生的。我们已经对它重新编辑了一下，以便使它能放在这里。

```
xtree(114) : warning C4786: 'std::_Tree<std::deque<std::
basic_string<char,std::char_traits<char>,std::allocator
<char>>,std::allocator<std::basic_string<char,std::
  ...“删去1420个字符”
allocator<char>>>>>>::iterator' : identifier was
truncated to '255' characters in the debug information
```

这个编译系统警告我们，它生成了一个长度为 1594个字符的变量名字(非常可观)，但是它只留下255个字符用在输出排错信息方面。并不是所有程序都能够保护自己，能对付这种罕见长度的字符串。

随机的输入(未必都是合法的)是另一种可能破坏程序的攻击方法。这也是“人们不会做这种事”的推理的一个逻辑延伸。例如，人们对某些商用编译程序用一些随机生成的合语法的程序做测试，这里采用的技巧是根据问题的规范 —— C语言标准 —— 驱动一个程序，产生合法的但又是希奇古怪的测试数据。

通过随机输入测试，考查的主要是程序的内部检查和防御机制，因为在这种情况下一般无法验证程序产生的输出是否正确。这种测试的目标主要是设法引起程序垮台，或者让它出现“不可能发生的情况”，而不是想发现直接的错误。这也是一种检查程序里错误检查代码能否工作的好方式。如果用的都是有意思的输入，大多数错误根本就不会发生，结果处理这些错误的代码无法得到执行，这就可能使程序错误隐藏在那些角落里。用随机输入测试有时也可能得不到什么回报：例如它可能发现了某些问题，可是这些问题在现实生活中出现的可能性微乎其微，以至都没有必要去考虑它们。

有些测试是针对明显的恶意输入进行的。安全性攻击经常使用极大的或者不合法的输入，设法引起对已有数据的覆盖。检查这方面的弱点是非常明智的。有些标准库函数在这类攻击下很容易受到伤害。例如，标准库函数gets没有提供限制输入行大小的方法，因此绝不要使用它，任何时候都改用fgets(buf, sizeof(buf), stdin)。不加限制的scanf(“%s”, buf)

China-pub.com 下载

调用对输入行长度也没有限制，应该改用带有明显长度限制的形式，如scanf(“%20s”,buf)。在第3.3节里，我们说明了对于一般的缓冲区大小，应该怎样处理好这个问题。

任何可能从程序外部接收信息的例程，无论是直接的或者间接的，都应该在使用有关数据之前对它们进行检验。下面是从某本教科书里抄来的一段程序，它要求由用户键盘输入读一个整数，在整数值太大时给用户提出一个警告。书中给出这个例子的目的是说明如何克服gets的问题，但这种解决办法有时并不奏效。

```
?    #define MAXNUM 10
?
?    int main(void)
?    {
?        char num[MAXNUM];
?
?        memset(num, 0, sizeof(num));
?        printf("Type a number: ");
?        gets(num);
?        if (num[MAXNUM-1] != 0)
?            printf("Number too big.\n");
?        /* ... */
?    }
```

如果输入包含10个或者更多的数字，就会有一个非零值覆盖掉数组 num最后的0，按说在gets返回后这个问题应该能被检查出来。不幸的是，这个检查是不够的。恶意攻击者可能提供一个非常长的输入串，它将覆盖掉数组后面的某些关键性数据，甚至覆盖掉函数调用的返回地址，使程序根本不返回到后面的 if语句，而可能是去做某些穷凶极恶的事情了。这种不加检查的输入确实是潜在的安全漏洞。

不要认为这不过是个无关紧要的教科书问题。在 1998年7月，几个最主要的电子邮件程序里都发现了这种形式的程序错误。《纽约时报》报道说：

> 这个安全漏洞是由所谓的“缓冲区溢出错误”引起的。按说程序员应该在他们的软件里包括一些代码，检查进入数据是否具有安全的类型，收到的数据单元是否具有正确长度。如果一个单元过长，它就可能越出“缓冲区”——即那种被设置好，用来存放数据的存储块。在这种情况下，该电子邮件程序就会崩溃，而一个恶意的程序员就可能欺骗计算机，使它运行放在某个位置的一个预谋的程序。

这也是在1988年著名的“Internet蠕虫”事件中被攻击的内容之一。

对这种向小数组里存储非常大的字符串的攻击方式，某些剖析 HTML形式的程序也可能是不设防的：

```
?    static char query[1024];
?
?    char *read_form(void)
?    {
?        int qsize;
?
?        qsize = atoi(getenv("CONTENT_LENGTH"));
?        fread(query, qsize, 1, stdin);
?        return query;
?    }
```

这段代码假定输入绝不会超过 1024字节长。就像gets一样，它对使其缓冲区溢出的攻击也没有设防。

China-pub.com
下载

另一些我们更熟悉的溢出同样可能引起麻烦。如果整数默默地溢出，其后果也可能是个灾难。考虑下面的存储分配：

```
?    char *p;
?    p = (char *) malloc(x * y * z);
```

假设x、y和z的乘积溢出，对malloc的调用有可能产生一个具有合理大小的数组。但是p[x]却可能引用到超出被分配区域的存储。假定int是16位，x、y和z的值都是41。这样x*y*z是68921，它在模2^{16}下就是3385。因此，对malloc的调用仅分配到3385个字节，任何超出这个值的下标引用都将越过界限。

类型间的转换是溢出的另一个原因，在这种情况下，甚至成功地捕捉到错误都不一定能解决问题。阿里亚娜五型火箭在1996年7月的初次试验中爆炸，是因为它由阿里亚娜四型那里继承来的导航系统没有经过很好的测试。新火箭飞得更快，结果导致了导航软件里某些变量具有更大的值。火箭发射后不久，有一次企图把64位浮点数转换到16位带符号整数的操作产生了溢出，这个错误其实被捕捉到了，但是，捕捉它的代码做出的选择是关闭这个子系统。这造成火箭转出航线并且爆炸。不幸的是，在这里出毛病的代码生成了惯性参考信息，这种信息本来只在起飞前有用，如果在发射时把这个功能关掉的话，也可能就不会出问题。

还有另一类更简单的情况，有时二进制输入会导致要求文本信息的程序崩溃，特别是如果程序假定输入的是7位的ASCII字符集。对那些要求文本输入的程序，将二进制输入(例如编译产生的程序)送给它做个试验也是有意义的，或者说是明智的。

好的测试实例经常能使用到许多不同的程序上。例如，任何读文件的程序都应该用空文件做测试；任何读文本文件的程序都应该用二进制文件做测试；任何读文本行的程序都应该用极长的行、空行和完全没有换行符号的输入做测试。在手头保存一批这种测试文件是很好的做法，这样，当你需要用它们测试任何程序时就不必重新建立了。另一种方式是写一个程序，在需要时用它生成这些文件。

当Steve Bourne写他的Unix外壳(即后来著名的Bourne外壳)时，他建立了一个目录，其中包含254个名字是一个字符的文件，每个字节值有一个对应文件，除了‘\0’和斜线符号之外(因为这两个字符不能出现在Unix文件名里)。他用这个目录做所有模式匹配和单词化方式的测试(这个目录本身当然也是利用程序构造的)。在此后的许多年里，这个目录一直是各种文件树遍历程序的灾星，经常能发现它们的毛病。

练习6-10 请设法建立一个文件，它能够击溃你最喜欢的文本编辑器、编译系统或者其他程序。

6.6 测试秘诀

有经验的测试者使用许多技术和技巧，以提高自己的工作效率。本节将介绍一些我们最喜欢的东西。

程序都应该检查数组的界限(如果语言本身不做这件事的话)，但是，如果数组本身的大小比典型输入大得多时，这种检查代码就常常测试不到。为演习这种检查，我们可以临时地把数组改为很小的值，这样做比建立大的测试实例更容易些。在第2章和第4章的CSV库中，我们对那里的数组增长代码都使用了这种技巧。实际上我们放了一个很小的初始值，这引起的附加的初始代价是微不足道的。

China-pub.com 下载

让散列函数返回某个常数值，使所有元素都跑到同一个散列桶里。这种做法能演习链的机制，它也指明了最坏情况下的性能。

写一个你自己的存储分配函数，有意让它早早地就失败，利用它测试在出现存储器耗尽错误时设法恢复系统的那些代码。下面的函数在调用 10次后就会返回NULL：

```
/* testmalloc: returns NULL after 10 calls */
void *testmalloc(size_t n)
{
    static int count = 0;
    if (++count > 10)
        return NULL;
    else
        return malloc(n);
}
```

在你提交代码的时候，应该关掉其中所有的测试限制，因为它们会影响程序性能。我们在使用某产品编译系统时遇到了性能问题，最后追踪到一个总返回 0的散列函数，原因是测试代码还在使用中。

把数组和变量初始化为某个可辨认的值，而不是总用默认的 0。这样，如果出现越界访问，或者取到了一个未初始化的变量值，你将更容易注意到它。常数 0xDEADBEEF在排错系统里很容易辨认，存储分配程序有时用这样的值，帮助捕捉未初始化的数据。

变动你的测试实例，特别是在用手工做小测试时。总使用同样东西很容易使人陷入某种常规，很可能忽略了其他的崩溃情况。

如果已经发现有错误存在，那么就不要继续设法去实现新特征或者再去测试已有的东西，因为那些错误有可能影响测试的结果。

测试输出中应该包括所有的参数设置，这可以使人容易准确地重做同样的测试。如果你的程序使用了随机数，应当提供方法，设置和打印开始的种子值，并使程序的行为与测试中是否使用随机性无关。应当保证能准确标识测试输入和对应的输出，这样才能理解和重新生成它们。

另一个做法也是很明智的，那就是提供一种方法来控制程序运行中输出的量和种类，附加的输出常能对测试有所帮助。

在不同的机器、编译系统和操作系统上做测试，每种组合都可能揭露出一些在其他情况下不能发现的错误。例如对于字节顺序的依赖性、对空指针的处理、对回车符和换行符的处理以及各种库或者头文件的特殊属性等等。在多种机器上测试还可以发现在集成程序的各个部件，以便发运过程中弄出的错误，可以揭示程序对开发环境的一些无意的依赖性，这是我们在第8章将要讨论的。

我们将在第7章讨论性能测试问题。

6.7 谁来测试

由程序实现者或其他可以接触源代码的人做的测试有时也被称为白箱测试 (这个术语与黑箱测试类似，但用的没有那么多。叫它“透明箱测试”可能更说明问题。在黑箱测试中测试者不知道部件的实现方式)。对自己的代码做测试是非常重要的，不要指望某种测试组织或者用户可以为你发现什么。但是，人很容易用自己已经非常仔细地做过测试来欺骗自己。所以，

China-pub.com
下载

你应该试着不考虑代码本身，仔细考虑最困难的测试实例，而不是那些容易的。这里引用 Don Knuth的一段话，他描述了自己如何为 TEX排版程序建立测试：“ 我设法使自己进入最卑劣的、极其令人讨厌的思想状态，写出自己能想到的最卑劣的测试代码，然后我再环顾四周，设法把它嵌入更加卑劣的几乎是污秽的结构之中 ”。做测试的原因就是要发现程序里的错误，而不是为了表明这个程序能够工作。所以，测试应该是恶毒的，如果发现了问题，那是你的方法有效的证明，根本不应该恐慌。

黑箱测试的测试者对代码的内部结构毫不知情，也无法触及。这样就可能发现另一类的错误，因为做测试的人对该在哪里做检查可能另有一些猜测。边界条件可能是开始做黑箱测试的好地方，大量的、邪恶的、非法的输入应该是随后的选择。当然你也应该测试常规的“ 大路货的 ”东西，测试程序的常规使用，验证其基本功能。

实际用户接踵而至。新用户往往能发现新错误，因为他们会以我们无法预知的方式来探测这个程序。在把一个程序发布到世界之前做这种测试是非常重要的，可惜的是许多程序在没有做好任何一种测试之前就发了货。软件的 Beta发布就是希望在它完成之前请一些真实用户来测试程序，但是， Beta发布绝不该被用做彻底测试的替代物。随着软件系统变得更大更复杂，开发进度表变得更短，不经过适当测试就发货的压力确实是在不断增加。

测试交互式程序是特别困难的，特别是如果它们还涉及到鼠标输入等。有些测试可以用脚本来做(脚本的特性依赖于语言、环境和其他类似东西)。交互式程序应该能够通过脚本控制，这样我们就可以用脚本模拟用户的行为，使测试可以通过程序完成。这方面的一种技术是捕捉真实用户的动作，并重新播放它；另一种技术是建立一个能表述事件序列和时间的脚本语言。

最后，还应该想一想如何测试所用的测试代码本身。在第 5章里，我们曾提到由表程序包的一个错误测试程序所造成的狼狈局面。如果一个回归测试集里感染了一个错误，能够造成的麻烦将是很长远的。如果一个测试集本身有毛病，由它得到的结果也就没有多少意义了。

6.8 测试马尔可夫程序

第3章的马尔可夫程序是非常复杂的，需要仔细地进行测试。这个程序要产生的是无意义的东西，因此很难分析其合法性。另外，我们还在几个语言里写了多个程序版本。在这里最麻烦的是程序输出是随机的，每次都不同。我们怎样才能把本章里学到的东西应用到这个程序的测试中呢？

第一个测试集由几个很小的文件组成，用于测试边界条件，目标是保证程序对只包含几个词的输入能正确产生输出。对于前缀长度为 2的情况，我们用了五个文件，它们分别包含(一行是一个文件)：

```
(空文件)
a
a b
a b c
a b c d
```

对于这里的每个文件，程序的输出都应该与输入完全相同。这些测试揭示出几个在表的初始化、生成程序的开始、结束等地方的“ 超出一个 ”错误。

第二项测试检验某些必须保持的特征。对于两词前缀的情况，一次运行中输出的每个词、

China-pub.com 下载

每个词对、以及每个三词序列都必然也出现在输入里。我们写了一个 Awk程序，用它把原始输入读进一个巨大的数组，构造出所有两个词和三个词的序列的数组，再把马尔可夫程序的输出读入另一个数组，然后对它们做比较。

```
# markov test:  check that all words, pairs, triples in
#   output ARGV[2] are in original input ARGV[1]
BEGIN {
    while (getline <ARGV[1] > 0)
        for (i = 1; i <= NF; i++) {
            wd[++nw] = $i    # input words
            single[$i]++
        }
    for (i = 1; i < nw; i++)
        pair[wd[i],wd[i+1]]++
    for (i = 1; i < nw-1; i++)
        triple[wd[i],wd[i+1],wd[i+2]]++

    while (getline <ARGV[2] > 0) {
        outwd[++ow] = $0    # output words
        if (!($0 in single))
            print "unexpected word", $0
    }
    for (i = 1; i < ow; i++)
        if (!((outwd[i],outwd[i+1]) in pair))
            print "unexpected pair", outwd[i], outwd[i+1]
    for (i = 1; i < ow-1; i++)
        if (!((outwd[i],outwd[i+1],outwd[i+2]) in triple))
            print "unexpected triple",
                outwd[i], outwd[i+1], outwd[i+2]
}
```

我们并不想建立一个效率很高的测试，而只是想使测试程序越简单越好。像这样把一个有10000个词的输出文件与一个42685个词的输入文件对照检查一次，需要6~7分钟时间，并不比有些马尔可夫程序版本生成这个文件用的时间长多少。对于应保持特征的检查捕捉到我们Java实现里的一个大错误：该程序有时复写掉散列表项，因为这个程序用的是引用，而不是做前缀的拷贝。

这个测试还显示了一个原理：验证输出的某种性质比建立这个输出本身要容易得多。例如，要检查一个文件是否已经排序就比把它排序容易得多。

第三步测试是统计性的，输入由下面的序列构成：

```
a b c a b c ... a b d ...
```

这里每十个abc有一个abd。如果随机选择部分工作得很好，那么在输出中 c的数目大约应该是d的十倍。我们用freq检验这个性质，没有问题。

在Java程序的一个早期版本里为每个后缀关联了一个计数器，统计测试说明它对每个 d大约生成20个c，比合理的情况多了一倍。经过令人头疼的努力，我们终于认识到 Java的随机数生成器不但产生正数，也产生负数。这里因子 2的出现是因为随机数值的范围是我们所期望的两倍，所以取模后得零的个数也就多了一倍，这就使得链表里第一个元素占了些便宜，而它恰好就是c。改正这个毛病只需在取模之前先求绝对值。没有这个测试，我们绝不可能发现这个错误，因为当时的输出用眼睛看起来非常好。

最后，我们给马尔可夫程序普通的英文文本，看着它产生出很漂亮的无意义的费话。当然，我们在程序开发的前期也运行过这种测试。但是，在程序处理正规输入时我们并没有放

China-pub.com
下载

弃测试，因为有些难对付的实例可能会出现在实际中。对简单的实例能够正确处理是很诱人的，难的实例也必须测试。自动的、系统化的测试是避免这种陷阱的最好方法。

所有这些测试都是机械化的。用一个脚本产生必需的输入数据，运行测试并且对它们计时，打印出反常的输出。这个脚本本身是可配置的，所以同一个测试能够应用到马尔可夫程序的各种版本上，每次我们对这些程序中的一个做了更改后，就重新运行所有测试，以保证所有东西都没出问题。

6.9 小结

你把开始的代码写得越好，它出现的错误也就越少，你也就越能相信所做过的测试是彻底的。在写代码的同时测试边界条件，这是去除大量可笑的小错误的最有效方法。系统化测试以一种有序方式设法探测潜在的麻烦位置。同样，毛病最可能出现在边界，这可以通过手工的或者程序的方式检查。自动进行测试是最理想的，用得越多越好，因为机器不会犯错误、不会疲劳、不会用臆想某些实际无法工作的东西能行来欺骗自己。回归测试检查一个程序是否能产生与它们过去相同的输出。在做了小改变之后就测试是一种好技术，能帮助我们将出现问题的范围局部化，因为新问题一般就出现在新代码里面。

对于测试，惟一的、最重要的规则就是必须做。

补充阅读

学习测试的一个方法就是研究最好的、能自由使用的软件中的实际例子。 Don Knuth在《软件：实践和经验》(Software: Practice and Experience19卷第7期pp607~685, 1989)的文章“The Errors of TEX”中描述了到当时为止在TEX排版程序中发现的每一个错误，还包含了许多对他的测试方法的讨论。为TEX构造的TRIP测试是完整测试集的一个绝佳的实例。 Perl也带有一个范围广泛的测试集，在它被编译安装到一个新系统以后，可以用来检验其正确性。这个测试集里还包括一些模块，如`MakeMaker`和`TestHarness`等，用来帮助测试Perl的扩充。

Jon Bentley在《美国计算机协会通信》(Communications of the ACM)上写过一系列文章，后来汇编成两本书，《程序设计精萃》(Programming Pearls)和《更多的程序设计精萃》(More Programming Pearls)，由Addison-Wesley公司分别于1986和1988年出版。文中常常谈到测试问题，特别是大量测试的组织和机械化问题。